API Testing Fundamentals

Postman to Pytest

modernQAcourse.com

THE MODERN QA ENGINEER SKILLSET

API Testing Fundamentals: Postman to Pytest

From Postman Exploration to Automated REST, GraphQL, and gRPC Pipelines

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

API Testing Fundamentals: Postman to Pytest

From Postman Exploration to Automated REST, GraphQL, and gRPC Pipelines

Book 14 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

About This Book

API testing is the highest-leverage testing activity in modern software engineering. A single API endpoint may serve the web application, mobile app, third-party integrations, and internal microservices simultaneously. A bug at the API layer propagates everywhere. An API test suite runs in seconds – no browser to launch, no DOM to render, no visual flakiness.

This book takes you from zero to production-ready API testing. You will begin with Postman for manual exploration, graduate to pytest for automated pipelines, master REST authentication and error handling, then extend your reach into GraphQL and gRPC. Every chapter includes real-world code examples, practical exercises at three difficulty levels, professional tips, and callouts for common mistakes.

By the end of this book, you will be able to design, implement, and maintain a comprehensive API test suite that runs in CI/CD, catches regressions before they reach users, and gives your team the confidence to deploy at any time.

Who This Book Is For

- **QA Engineers** transitioning from manual to automated API testing
- **Software Developers** who want to build robust API test coverage
- **SDET Candidates** preparing for technical interviews that focus on API quality
- **DevOps Engineers** integrating API test suites into CI/CD pipelines

- **Technical Leads** establishing API testing standards for their teams

Prerequisites

- Basic understanding of HTTP (methods, status codes, headers)
- Familiarity with JSON data format
- Python 3.8+ installed on your machine
- Postman desktop application installed (free tier is sufficient)
- Basic command-line comfort (running scripts, installing packages)
- A text editor or IDE (VS Code, PyCharm, or similar)

Recommended Setup

```
# Create a virtual environment for the exercises
python -m venv api-testing-env
source api-testing-env/bin/activate  # macOS/Linux
# api-testing-env\Scripts\activate   # Windows

# Install core dependencies
pip install requests pytest pytest-xdist jsonschema grpcio grpcio-tools

# Install Newman (requires Node.js)
npm install -g newman newman-reporter-htmlextra
```

Table of Contents

- Chapter 8: API Versioning – Backward Compatibility and Deprecation
- Chapter 9: Capstone Project – Building a Production API Test Suite
- Appendix A: 50 API Testing Exercises (Complete Reference)
- Appendix B: Glossary

CHAPTER 1

Postman Mastery – API Exploration and Manual Testing

Before you automate, understand the API. Postman is the universal tool for manual API exploration – it lets you send requests, inspect responses, chain calls together, and build a mental model of how the API behaves before writing a single line of test code.

This chapter covers everything you need to become proficient with Postman: importing specs, organizing collections, writing test scripts, chaining requests, data-driven testing, and knowing when to move beyond Postman to code-based automation.

1.1 Getting Started with Postman

Importing an API Specification

Most modern APIs provide an OpenAPI (Swagger) specification. Importing it into Postman auto-generates a collection with every endpoint, request body schema, and example parameters.

```
File > Import > OpenAPI/Swagger URL or file
```

This gives you an instant starting point: every endpoint is documented with expected parameters and response schemas. If the API does not provide an OpenAPI spec, you can create requests manually or import from a cURL command.

> **PRO TIP**: Always start by importing the OpenAPI spec rather than creating requests manually. This saves hours of setup time and ensures you do not miss any endpoints. Many teams maintain their OpenAPI spec alongside their code – ask your developers for the latest version.

Setting Up Environments

Environments hold variables that switch between dev, staging, and production without modifying requests. This is one of Postman's most powerful features.

Variable	Dev	Staging	Production
`{{base_url}}`	`http://localhost:3000`	`https://api.staging.example.com`	`https://api.example.com`
`{{auth_token}}`	(dev token)	(staging token)	(read-only prod token)
`{{timeout}}`	5000	10000	10000

Switch environments with the dropdown in the top-right corner. All requests using `{{base_url}}` automatically point to the correct server.

> **COMMON MISTAKE**: Hardcoding URLs directly in requests instead of using environment variables. When you inevitably need to switch environments, you will have to update every single request manually. Always use `{{base_url}}` from the start.

Organizing Collections

Structure collections to mirror the API structure. This makes it easy to find requests, run subsets of tests, and onboard new team members.

```
My API Collection/
  Auth/
    POST Login
    POST Refresh Token
    POST Logout
  Users/
    GET List Users
    GET Get User by ID
    POST Create User
    PUT Update User
    DELETE Delete User
  Orders/
    GET List Orders
    POST Create Order
    GET Get Order by ID
```

Group related endpoints into folders. Name requests clearly with the HTTP method prefix. Order them in the sequence you would typically execute them (login first, then CRUD operations).

1.2 Writing Test Scripts

Postman's test scripts run after each request, validating responses inline. These scripts use JavaScript and the Chai assertion library through Postman's `pm` API.

Basic Assertions

```
// Status code verification
pm.test("Status is 200", () => {
    pm.response.to.have.status(200);
});

// Response body field validation
pm.test("Response has user ID", () => {
    const data = pm.response.json();
    pm.expect(data.id).to.be.a("number");
    pm.expect(data.email).to.be.a("string");
    pm.expect(data).to.not.have.property("password");
});

// Response time validation
pm.test("Response time < 500ms", () => {
    pm.expect(pm.response.responseTime).to.be.below(500);
});

// Header verification
pm.test("Content-Type is JSON", () => {
    pm.response.to.have.header(
        "Content-Type", "application/json; charset=utf-8"
    );
});
```

Chaining Requests with Environment Variables

Extract data from one request and use it in the next. This is essential for testing authenticated workflows and CRUD sequences.

```
// In POST /auth/login Tests tab:
pm.test("Save auth token", () => {
    const token = pm.response.json().access_token;
    pm.environment.set("auth_token", token);
});

// In subsequent requests, use {{auth_token}} in the Authorization header:
// Bearer {{auth_token}}
```

This pattern is fundamental: login once, store the token, and reuse it across all subsequent requests. When the token expires, simply re-run the login request.

Dynamic Variables

Postman provides built-in dynamic variables for generating test data on the fly:

```
{{$randomEmail}}      -> random email address
{{$randomFullName}}   -> random name
{{$randomUUID}}       -> UUID v4
{{$timestamp}}        -> current Unix timestamp
{{$randomInt}}        -> random integer
```

These are useful for creating unique test data without hardcoding values that might cause conflicts.

1.3 Pre-Request Scripts

Pre-request scripts run before the request is sent – useful for generating dynamic data, calculating signatures, or setting up authentication.

```
// Generate a unique email for each test run
const timestamp = Date.now();
pm.environment.set("test_email", `testuser_${timestamp}@test.com`);

// Calculate HMAC signature for signed requests
const CryptoJS = require('crypto-js');
const secret = pm.environment.get("api_secret");
const payload = JSON.stringify(pm.request.body.raw);
const signature = CryptoJS.HmacSHA256(payload, secret).toString();
pm.request.headers.add({ key: "X-Signature", value: signature });
```

> **PRO TIP**: Use pre-request scripts to generate timestamps, unique identifiers, and cryptographic signatures. This ensures each test run uses fresh data and prevents test interference from stale values.

1.4 Collection Runner and Data-Driven Testing

The Collection Runner executes all requests in a collection sequentially, with test assertions evaluated at each step.

Running a Collection

1. Click “Runner” in Postman
2. Select the collection (or a specific folder)
3. Choose the environment
4. Set iterations (run the entire collection N times)
5. Upload a data file for data-driven testing

Data-Driven Testing with CSV Files

Create a CSV or JSON file with test data. Each row becomes one iteration of the collection run.

```
email,password,expected_status
valid@test.com,ValidPass123!,200
invalid@test.com,wrong,401
,ValidPass123!,400
valid@test.com,,400
```

In the request body, reference columns: `{{email}}`, `{{password}}`
In the test script:

```
pm.test(`Login with ${pm.iterationData.get("email")} returns ` +
        `${pm.iterationData.get("expected_status")}`, () => {
    pm.response.to.have.status(
        parseInt(pm.iterationData.get("expected_status"))
    );
});
```

This approach lets you test dozens of input combinations without duplicating requests.

1.5 Monitors and Workspaces

Monitors

Schedule collections to run automatically (for example, every hour) and receive alerts when tests fail. Monitors are useful for production health checks and continuous API monitoring.

Workspaces

Share collections across team members. Changes sync automatically. Use workspaces for team API documentation and shared test collections. Workspaces are particularly valuable when onboarding new team members – they can immediately access all existing API tests and documentation.

1.6 Understanding Postman's Limitations

Limitation	Impact	Alternative
Not ideal for CI/CD	Requires Newman for command-line execution	Use Newman or pytest directly
Collections become unwieldy at scale	Hard to manage 500+ requests	Use code-based tests (pytest)
Limited programming model	JavaScript only, limited libraries	Full language ecosystem in pytest
Version control	JSON export is hard to diff	Code-based tests in Git
Complex assertions	Difficult to build reusable assertion libraries	Python/TypeScript assertion libraries

Postman excels at exploration and prototyping. For production CI/CD pipelines, transition to code-based automation.

> **COMMON MISTAKE**: Trying to build your entire test automation strategy on Postman alone. Postman is an exploration and prototyping tool. For maintainable, scalable test suites, you need code-based automation.

1.7 Postman Collection Export Format

When you export a Postman collection, it produces a JSON file. Understanding this format is important for integrating with Newman and version control.

```
{
    "info": {
        "name": "My API Tests",
        "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
    },
    "item": [
        {
            "name": "Auth",
            "item": [
                {
                    "name": "POST Login",
                    "request": {
                        "method": "POST",
                        "header": [],
                        "body": {
                            "mode": "raw",
                            "raw": "{\"email\": \"{{email}}\", \"password\": \"{{password}}\"}"
                        },
                        "url": {
                            "raw": "{{base_url}}/auth/login",
                            "host": ["{{base_url}}"],
                            "path": ["auth", "login"]
                        }
                    },
                    "event": [
                        {
                            "listen": "test",
                            "script": {
                                "exec": [
                                    "pm.test('Status is 200', () => {",
                                    "    pm.response.to.have.status(200);",
                                    "});"
                                ]
                            }
                        }
                    ]
                }
            ]
        }
    ]
}
```

Chapter 1 Exercises

Beginner

Exercise 1: Import the JSONPlaceholder API (`https://jsonplaceholder.typicode.co` into Postman. Create requests for `GET /posts`, `GET /posts/1`, and `POST /posts`. Write test scripts that verify the status code for each request.

Exercise 2: Create two environments in Postman – "JSONPlaceholder" and "Local Dev." Set the `base_url` variable appropriately for each. Verify that switching environments changes the target server for all requests.

Intermediate

Exercise 3: Build a data-driven test for the JSONPlaceholder `POST /posts` endpoint. Create a CSV file with five different post titles and bodies. Run the collection with the CSV file and verify that each iteration returns status 201.

Exercise 4: Chain three requests together: (1) Create a post, (2) extract the post ID from the response and save it to an environment variable, (3) use the saved ID to fetch that specific post with `GET /posts/{{post_id}}`. Write assertions for each step.

Advanced

Exercise 5: Write a pre-request script that generates an HMAC-SHA256 signature from the request body using a secret stored in an environment variable. Attach the signature as a custom header (`X-Signature`). Then write a test script that verifies the response acknowledges the signature.

Career Translation

Resume phrasing

- Designed and maintained Postman collections covering 100% of API endpoints, reducing manual regression testing time by 60% during sprint cycles.
- Built data-driven API test suites in Postman with parameterized CSV inputs, enabling non-technical stakeholders to contribute test scenarios.
- Established Postman workspace standards (environment management, collection organization, pre-request scripts) adopted across a 15-person QA team.

Cover letter framing

API testing begins with understanding the system under test, and Postman is where that understanding starts. I use Postman not as a crutch but as an exploration tool – importing OpenAPI specs, chaining requests to map authentication flows, and building data-driven tests to validate boundary conditions before investing in automation. This systematic exploration

phase consistently reduces downstream automation rework by catching spec ambiguities early.

Interview framing

"I treat Postman as my API reconnaissance tool. Before I write a single line of automated code, I import the OpenAPI spec, set up environment variables for each deployment target, and manually walk through every critical workflow – login, CRUD, error paths. I chain requests using environment variables so I can validate multi-step flows like token refresh cycles. Once I understand the API's behavior and edge cases, I export the collection for Newman CI runs while transitioning the long-term suite to code-based automation. The trade-off is clear: Postman gives you speed of exploration, but code gives you maintainability at scale."

What not to say

- "I just click Send and check the response." – Shows no systematic approach; interviewers want to see structured thinking.
- "I use Postman for all my test automation." – Signals you do not understand when to transition to code-based tools.
- "I hardcode URLs and tokens directly in my requests." – Reveals poor environment management practices that break in real teams.
- "I do not use pre-request scripts because they are too complicated." – Suggests an unwillingness to handle real-world complexity like HMAC signing or dynamic data generation.
- "I only test the happy path and assume errors are handled." – Missing negative test coverage is a red flag for any senior role.

Interview Depth Check

Question 1

Prompt: You inherit a Postman collection with 200+ requests, no environment variables, hardcoded tokens everywhere, and no folder organization. The team wants to run it in CI within two weeks. Walk me through your remediation plan.

What a strong answer should cover:

- Prioritization: identify the critical paths first (auth, core CRUD) rather than trying to fix everything at once
- Environment variable extraction as the first step (base_url, tokens, dynamic IDs)
- Folder restructuring to mirror the API domain model
- Adding pre-request scripts for token management instead of hard-coded values
- Newman integration for CI with a plan to eventually migrate high-value tests to pytest

Example answer:

- "First, I would audit the collection to identify which requests are actually used and which are stale. I would group them by domain (auth, users, orders) and delete dead requests."
- "Next, I would extract all hardcoded URLs into a `base_url` environment variable and create separate environments for dev, staging, and CI. Tokens get replaced with a login pre-request script that runs once and stores the token."
- "For CI within two weeks, I would export the cleaned collection and run it via Newman in GitHub Actions. The long-term plan would be migrating critical test paths to pytest, but Newman gets us CI coverage immediately."
- "The trade-off is that Newman collections are harder to maintain than code, so I would set a clear timeline for the pytest migration – probably the following sprint."

Question 2

Prompt: A developer tells you their API does not need an OpenAPI spec because "the code is the documentation." How do you push back, and what testing risks does the absence of a spec create?

What a strong answer should cover:

- Concrete testing impact: no auto-generated Postman collections, no schema validation, no contract testing
- Team-wide costs: every new tester has to reverse-engineer the API by reading code or asking developers
- Drift risk: without a spec, the actual behavior and the intended behavior diverge silently
- Pragmatic compromise: even a minimal spec is better than none

Example answer:

- "Without an OpenAPI spec, I cannot auto-generate Postman collections, so every endpoint has to be discovered manually – which is slow and error-prone."
- "More importantly, I lose the ability to do schema validation in tests. Instead of validating responses against a contract, I am guessing at what fields should be present. When the developer adds or removes a field, my tests do not catch it unless I manually update expectations."
- "I would propose a compromise: let me generate a spec from the existing code using a tool like swagger-autogen or FastAPI's built-in spec generation. That gives us a starting point without requiring the developer to write it from scratch."

Question 3

Prompt: You are building a data-driven test in Postman with a CSV file containing 500 rows of test data. The collection run takes 45 minutes. How do you optimize this?

What a strong answer should cover:

- Analysis first: identify which requests are slow (network latency vs. test setup)
- Reducing unnecessary requests (skip redundant auth calls by using collection-level pre-request scripts)
- Parallelization strategies (splitting the CSV, running multiple Newman instances)

- Questioning whether all 500 rows are necessary or if equivalence partitioning can reduce to 50 representative cases

Example answer:

- "First, I would check if each iteration is re-authenticating. If so, I would move the login to a collection-level pre-request script that runs once and stores the token, not per-iteration."
- "Second, I would apply equivalence partitioning to the CSV. If 400 of the 500 rows test valid inputs with minor variations, I would reduce those to 20 representative cases. The 100 boundary and error cases likely stay."
- "Third, I would split the remaining CSV into chunks and run multiple Newman instances in parallel using CI matrix jobs. Four parallel runners would cut the time to roughly 25% of the original."
- "Finally, if 45 minutes is still too long, that is a signal to migrate to pytest with pytest-xdist, which gives true parallel execution with better resource management."

About This Sample

You have just read **Chapter 1 of API Testing Fundamentals: Postman to Pytest** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-14-api-testing-fundamentals/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.